Implementing Polymorphism in C:*

A Survey of Idioms for Type-Based and Signature-Based Dispatch

Remo Dentato

Rootnode Distillery

rdentato@rootnodedistillery.eu

12 December 2025

Abstract

Polymorphism, the ability to write functions whose behavior depends on the type and number of their arguments, is absent from C, yet essential for crafting usable library APIs. Without it, libraries expose numerous near-identical functions differing only in name and signature, burdening users with unnecessary complexity. This paper surveys techniques that enable polymorphic interfaces in C: five for type-based dispatch (tagged structures, function-pointer tables, tagged pointers, C11's `_Generic`, and the `container_of` idiom) and three for signature-based dispatch (default substitution, argument selection, and argument counting). We analyze each approach with respect to type safety, performance, memory overhead, and portability, with attention to strict aliasing rules and `_Generic` limitations. By applying these idioms, library authors can present consistent, intuitive APIs that allow users to focus on higher-level problem-solving rather than interface mechanics.

Keywords: Polymorphism, C programming, generic programming, type dispatch, variadic macros, `_Generic`, `container_of`, library design

1 Introduction

Polymorphism enables software developers to write code that operates uniformly across different data types, reducing redundancy and improving maintainability. In object-oriented

*A preliminary version of this article appeared on DEV.to at `https://dev.to/rdentato/polymorphic-c-4kk2`.

languages such as C++ and Java, polymorphism is typically achieved through method overloading (ad-hoc polymorphism) and inheritance-based virtual dispatch (subtype polymorphism). Functional languages like Haskell provide parametric polymorphism through type classes, while Rust combines traits with monomorphization for zero-cost abstractions.

The C programming language, designed with simplicity and direct hardware access as primary goals, provides no built-in polymorphism mechanisms. This absence is particularly burdensome for *library developers*, programmers who create reusable components for consumption by other programmers. Without polymorphism, library APIs become cluttered with nearly identical functions distinguished only by name and signature:

```
result_t mix_int_str(int x, char *y);
result_t mix_uint_str(unsigned x, char *y);
result_t mix_str_float(char *x, float *y);
```

Such proliferation of function names impedes API usability and increases cognitive load. Library users must memorize dozens of variants, select the correct one for each call site, and update their code when types change. This friction distracts from the higher-level problems they are trying to solve.

The techniques presented in this paper address this challenge directly. By applying polymorphic idioms, library authors can expose a single, intuitive entry point, `mix(x, y)`, that automatically dispatches to the appropriate implementation based on argument types or count. The underlying complexity remains encapsulated within the library, invisible to its users. This separation of concerns allows application programmers to focus on domain logic rather than interface mechanics, while library maintainers retain full control over implementation details.

This paper demonstrates that polymorphic interfaces are achievable in C through careful application of several idioms. We organize our presentation into two main categories: *type-based dispatch* (Section 2), where behavior depends on argument types, and *signature-based dispatch* (Section 3), where behavior depends on the number of arguments. Section 4 provides a comparative analysis, Section 5 contextualizes these techniques relative to other systems languages, Section 6 offers practical recommendations, and Section 7 concludes.

2 Type-Based Polymorphism

We consider a running example of graphical objects (points, rectangles, circles) supporting operations such as `scale()` and `translate()`. Figure 1 illustrates the memory layout differences among the primary approaches.

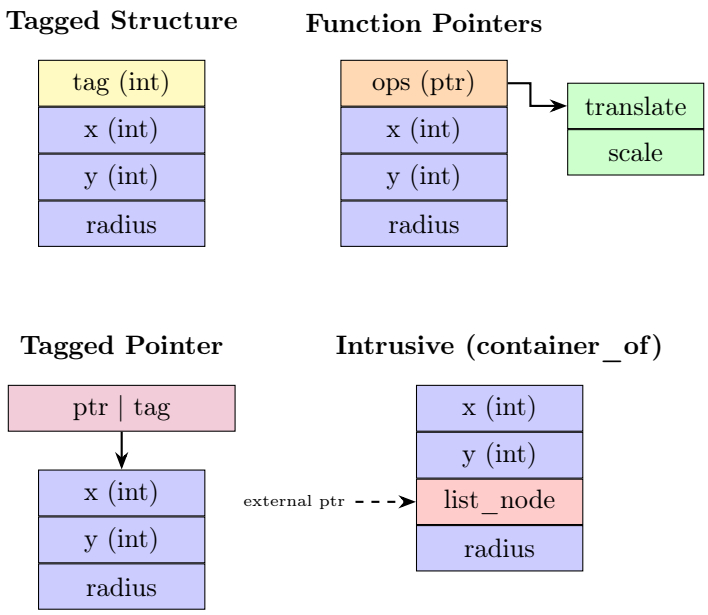

Figure 1: Memory layout comparison of polymorphism techniques. Blue regions represent object data; colored regions represent polymorphism overhead. Tagged structures embed a type identifier; function-pointer approaches store a pointer to a shared vtable; tagged pointers encode type in unused pointer bits; intrusive structures embed a base structure within the object.

2.1 Void Pointers with Tagged Structures

The simplest approach stores a type tag within each structure, enabling runtime inspection and dispatch.

```
#define OBJ_POINT  0
#define OBJ_RECT   1
#define OBJ_CIRCLE 2

typedef struct { int tag; } GraphicObject;

typedef struct {
    int tag;  // OBJ_POINT
    int x, y;
} Point;

typedef struct {
    int tag;  // OBJ_CIRCLE
    int x, y;
    float radius;
} Circle;
```

A union type ensures that accessing the common initial sequence (the `tag` field) is well-defined per the C11 standard [1], specifically §6.5.2.3, which states that if a union contains several structures sharing a common initial sequence, inspecting the common initial part of any of them is permitted:

```
typedef union {
    GraphicObject object;
    Point point;
    Circle circle;
} ObjectType;

static inline int get_tag(void *p) {
    return ((ObjectType *)p)->object.tag;
}
```

Dispatch functions examine the tag via a `switch` statement:

```
void scale(void *obj, float sx, float sy) {
    switch (get_tag(obj)) {
        case OBJ_POINT:  break;
        case OBJ_RECT:   scale_rect(obj, sx, sy); break;
        case OBJ_CIRCLE: scale_circle(obj, sx, sy); break;
    }
}
```

Advantages: Simplicity; uniform treatment of all pointer types.

Disadvantages: No compile-time type checking (passing `stdout` compiles without error); runtime dispatch overhead; per-object memory overhead for tags.

2.1.1 Strict Aliasing Considerations

The C11 standard's strict aliasing rules ([1], §6.5, paragraph 7) specify that an object shall have its stored value accessed only by an lvalue expression of a compatible type. Accessing a `Circle` through a `GraphicObject*` is well-defined *only* when using the union-based access pattern shown above.

Casting arbitrary `void*` pointers without proper provenance can lead to undefined behavior under aggressive compiler optimization. For example:

```
void dangerous_access(void *p) {
    // UB if p doesn't actually point to a GraphicObject
    GraphicObject *g = (GraphicObject *)p;
    int t = g->tag;  // Strict aliasing violation possible
}
```

In practice, many legacy C projects compile with `-fno-strict-aliasing` (GCC/Clang) to disable these optimizations. However, this is a non-portable workaround. The safest approach is to consistently use union-based access or ensure that all pointers originate from properly typed allocations.

2.2 Function-Pointer Tables

This approach embeds operation pointers within objects, analogous to C++ virtual tables (vtables).

```
typedef struct {
    void (*translate)(void *self, int dx, int dy);
    void (*scale)(void *self, float sx, float sy);
} GraphicOps;

typedef struct {
    const GraphicOps *ops;
    int x, y;
    float radius;
} Circle;

static const GraphicOps circle_ops = {
    .translate = (void(*)(void*,int,int))translate_circle,
    .scale = (void(*)(void*,float,float))scale_circle,
};
```

Dispatch occurs through indirect function calls:

```
#define scale(obj,sx,sy) ((obj)->ops->scale(obj,sx,sy))
```

Advantages: Single indirect call (no switch); compile-time type safety for the object pointer.

Disadvantages: Pointer-sized overhead per object; macro wrappers double-evaluate the `obj` argument (beware side effects per [1], §6.10.3); initialization boilerplate.

This approach directly mirrors C++'s vtable implementation and Rust's `dyn Trait` fat pointers, though in those languages the compiler manages vtable construction automatically.

2.3 Tagged Pointers

Modern architectures enforce memory alignment, leaving low-order pointer bits unused. These bits can encode type information without per-object overhead. The C11 standard provides the `max_align_t` type ([1], §7.19) and the `alignof` operator ([1], §6.5.3.4) to portably determine alignment requirements:

```
#include <stdint.h>
#include <stdalign.h>

#define MAX_TAGS alignof(max_align_t)
#define TAG_MASK ((uintptr_t)(MAX_TAGS-1))
#define PTR_MASK (~TAG_MASK)

// Verify alignment is sufficient and is a power of two
_Static_assert(alignof(max_align_t) >= 4,
    "Need at least 4-byte alignment for tag bits");
_Static_assert((alignof(max_align_t) & (alignof(max_align_t)-1)) == 0,
    "Alignment must be a power of two for masking");

typedef struct { uintptr_t ptr; } TaggedPtr;

static inline TaggedPtr tag_ptr(void *p, int tag) {
    return (TaggedPtr){(uintptr_t)p | tag};
}

static inline void *untag_ptr(TaggedPtr tp) {
    return (void *)(tp.ptr & PTR_MASK);
}

static inline int get_tag(TaggedPtr tp) {
    return (int)(tp.ptr & TAG_MASK);
}
```

Objects must be allocated with appropriate alignment using `aligned_alloc()`, introduced in C11 ([1], §7.22.3.1):

```c
TaggedPtr new_circle(float r) {
    Circle *c = aligned_alloc(MAX_TAGS, sizeof *c);
    if (!c) return (TaggedPtr){0};
    c->radius = r;
    return tag_ptr(c, OBJ_CIRCLE);
}
```

Advantages: Zero per-object overhead; distinct pointer type prevents accidental misuse.

Disadvantages: Limited tag space (typically 3–4 bits); runtime switch dispatch; portability concerns on non-standard architectures. The `uintptr_t` type ([1], §7.20.1.4) is essential for these conversions, though the standard notes that it is optional.

Platform Variability. On most 64-bit systems, `alignof(max_align_t)` is 16 bytes, providing 4 tag bits. However, on some 32-bit embedded systems, alignment may be only 4 or 8 bytes. The `_Static_assert` declarations above ensure compile-time failure if assumptions are violated.

Extended Tagging. On architectures using 48-bit virtual addresses, the upper 16 bits may store tags, permitting up to 2^{16} distinct types. This technique sacrifices portability for expanded type capacity and relies on implementation-defined behavior when converting between integers and pointers ([1], §6.3.2.3).

2.4 Compile-Time Dispatch with `_Generic`

C11 introduced the `_Generic` keyword ([1], §6.5.1.1), enabling compile-time selection based on expression type. This mechanism provides a form of ad-hoc polymorphism resolved entirely at compile time:

```c
#define scale(obj, sx, sy) \
    _Generic((obj), \
        Circle*:    scale_circle, \
        Rectangle*: scale_rect, \
        Point*:     scale_point \
    )(obj, sx, sy)
```

Advantages: Zero runtime overhead; full compile-time type safety.

Disadvantages: Cannot dispatch on runtime type; verbose macro definitions; cannot iterate heterogeneous collections; subject to the type compatibility rules of `_Generic` ([1], §6.5.1.1), which may require careful function signature design.

2.4.1 The Exact-Match Requirement

A critical limitation of `_Generic` is that it requires *exact* type matching. It does not perform implicit conversions, qualifier decay, or promotion. This leads to surprising failures:

```
const Circle *cc = get_circle();
scale(cc, 2.0f, 2.0f);  // ERROR: no match for 'const Circle*'
```

The `const Circle*` type does not match the `Circle*` association. Similarly, array-to-pointer decay and integer promotions are not applied to the controlling expression.

Workarounds. Two common approaches address this limitation:

1. **Explicit `const` entries:** List all qualifier variants explicitly.

```
#define scale(obj, sx, sy) \
    _Generic((obj), \
        Circle*:       scale_circle, \
        const Circle*: scale_circle, \
        /* ... */ \
    )(obj, sx, sy)
```

2. **Use `default` association:** Provide a fallback, though this sacrifices some type safety.

```
#define scale(obj, sx, sy) \
    _Generic((obj), \
        Circle*:    scale_circle, \
        Rectangle*: scale_rect, \
        default:    scale_generic \
    )(obj, sx, sy)
```

Neither solution is ideal; the first is verbose, while the second may mask errors. Library designers must weigh these trade-offs carefully.

2.5 Hybrid: Type-Safe Void Pointers

Combining `_Generic` with tagged structures provides compile-time checking while permitting runtime dispatch:

```
#define scale(obj, sx, sy) \
    _Generic((obj), \
        GraphicObject*: scale_dispatch, \
        Point*:         scale_dispatch, \
        Circle*:        scale_dispatch \
    )(obj, sx, sy)
```

This ensures only valid pointer types reach `scale_dispatch()`, eliminating the `scale(stdout, ...)` vulnerability while retaining runtime flexibility.

2.6 Intrusive Structures and the `container_of` Idiom

Perhaps the most "practical" form of polymorphism in systems C is the *intrusive* approach, popularized by the Linux kernel. Rather than embedding a pointer to a vtable or a type tag, this technique embeds a common structure *within* each object type, then recovers the containing object via pointer arithmetic.

2.6.1 The `container_of` Macro

The key primitive is the `container_of` macro, which computes the address of an enclosing structure given a pointer to one of its members. This relies on `offsetof` ([1], §7.19):

```c
#include <stddef.h>

#define container_of(ptr, type, member) \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

2.6.2 Application to Polymorphism

Consider a generic linked-list node that can be embedded in any structure:

```c
typedef struct list_node {
    struct list_node *next;
    struct list_node *prev;
} list_node;

typedef struct {
    int x, y;
    list_node node;  // Embedded, not pointed-to
    float radius;
} Circle;

typedef struct {
    int x, y;
    list_node node;
    int width, height;
} Rectangle;
```

A generic list iteration function operates on `list_node*` pointers:

```c
void list_foreach(list_node *head, void (*fn)(list_node *)) {
    for (list_node *n = head; n != NULL; n = n->next)
        fn(n);
}
```

Within the callback, the containing object is recovered:

```c
void print_circle(list_node *n) {
    Circle *c = container_of(n, Circle, node);
    printf("Circle at (%d,%d) r=%.2f\n", c->x, c->y, c->radius);
}
```

2.6.3 Multiple Interface Participation

A key advantage of intrusive structures is that a single object can participate in *multiple* polymorphic interfaces simultaneously by embedding multiple base structures:

```c
typedef struct {
    int x, y;
    list_node scene_list;    // For scene graph iteration
    list_node render_queue;  // For render ordering
    float radius;
} Circle;
```

This is analogous to multiple inheritance in C++, but without vtable pointer overhead for each interface.

Advantages: No per-object pointer overhead; supports multiple simultaneous interfaces; widely used in production systems (Linux kernel, GLib, etc.); cache-friendly due to data locality.

Disadvantages: Requires manual offset arithmetic; type safety depends on programmer discipline; the embedded structure's position affects memory layout; cannot be used with opaque types.

3 Signature-Based Polymorphism

Beyond type variation, polymorphic APIs often require handling different argument counts. C's variadic macros, standardized in C99 and refined in C11 ([1], §6.10.3), provide the foundation for these techniques. The `__VA_ARGS__` identifier ([1], §6.10.3.1) enables macros to accept and forward variable argument lists.

3.1 Default Argument Substitution

When optional trailing arguments default to zero, a simple idiom suffices:

```c
#define move(s,...) move_f(s, __VA_ARGS__ + 0)
void move_f(int steps, int offset);
```

Invoked as `move(10)`, this expands to `move_f(10, +0)`. For pointer arguments, `_Generic` ensures type safety. Note that while the integer constant expression `0` converted to `void*`

is a null pointer constant ([1], §6.3.2.3), its *bit representation* may not be all-zero on some architectures. Therefore, explicit `NULL` casts are advisable:

```
#define transfer(i,...) transfer_g(i, __VA_ARGS__ +0)
#define transfer_g(i,a) _Generic((a), \
    info_t*: transfer_f(i,(info_t*)(a)), \
    int:     transfer_f(i,(info_t*)NULL))
```

3.2 Argument Selection Macros

For multiple optional arguments, selection macros extract positional arguments:

```
#define arg_0(x,...)         x
#define arg_1(_0,x,...)      x
#define arg_2(_0,_1,x,...)   x

#define move(...) move_f( \
    arg_0(__VA_ARGS__), \
    arg_1(__VA_ARGS__, 0), \
    arg_2(__VA_ARGS__, 'N', 'N'))
```

The repeated default values ensure correct expansion regardless of argument count. The comma operator semantics ([1], §6.5.17) and macro replacement rules ([1], §6.10.3.1) govern these expansions. When combined with `_Generic`, this permits "skipping" intermediate arguments when types are distinguishable.

3.3 Argument Counting

The most flexible approach counts arguments and dispatches to specialized implementations:

```
#define ARG_CNT(_1,_2,_3,_4,N,...) N
#define ARG_COUNT(...) ARG_CNT(__VA_ARGS__,4,3,2,1,0)
#define ARG_JOIN(x,y)  x ## y
#define ARG_VRG(x,y)   ARG_JOIN(x,y)

#define move(...) ARG_VRG(move_,ARG_COUNT(__VA_ARGS__))(__VA_ARGS__)

#define move_1(s)     move_f(s, 0, 'N')
#define move_2(s,t)   move_f(s, t, 'N')
#define move_3(s,t,d) move_f(s, t, d)
```

This technique enables radically different behavior per arity, including mixing getters and setters within a single identifier. The token-pasting operator `##` ([1], §6.10.3.3) is essential for constructing the dispatched function names.

3.4 Zero-Argument Handling

The preceding techniques assume at least one argument. C23's `__VA_OPT__` keyword addresses this limitation, though widespread compiler support remains forthcoming. Until then, handling zero arguments in a strictly C11-compliant manner requires complex workarounds that may compromise code clarity.

4 Comparative Analysis

Table 1 summarizes the trade-offs among type-dispatch techniques.

Table 1: Comparison of Type-Dispatch Techniques

Technique	**Safety**	**Runtime**	**Memory**	**Portability**	**Multi-iface**
Tagged structs	Low	Switch	+int	High	No
Vtables	Medium	Indirect	+ptr	High	No
Tagged ptrs	Medium	Switch	None	Medium	No
`_Generic`	High	None	None	High	N/A
Intrusive	Medium	None*	None	High	Yes

*Dispatch overhead depends on callback implementation.

For signature-based dispatch, argument counting provides maximum flexibility at the cost of macro complexity, while default substitution offers simplicity for limited use cases. A complete implementation of signature-based macros that can also handle zero arguments is presented in Appendix A.

5 Context: Polymorphism in Systems Languages

The techniques presented here occupy a specific niche in the design space of polymorphism mechanisms. C++'s vtables are directly analogous to our function-pointer tables (Section 2.2), but compiler-managed. Rust's `dyn Trait` uses fat pointers combining data and vtable references, conceptually similar, but with compile-time trait bound verification. Rust's `enum` types with pattern matching provide a type-safe alternative to C's manual tag checking, with exhaustiveness verification.

The `container_of` idiom (Section 2.6) has no direct analogue in C++ or Rust, where inheritance and trait objects provide cleaner abstractions. However, it remains valuable in C for its zero-overhead multiple-interface capability.

C's `_Generic` provides compile-time dispatch similar to C++ function overloading, but with exact-match semantics rather than overload resolution. Rust achieves similar goals through monomorphization of generic functions, with stronger type inference.

The key distinction is that C requires explicit programmer effort for what other languages automate, but this effort yields precise control over memory layout and dispatch mechanisms, critical for systems programming where every byte and cycle matters.

6 Practical Recommendations

Based on our analysis, we offer the following guidelines:

1. **Prefer `_Generic` for static polymorphism.** When argument types are known at compile time, `_Generic` ([1], §6.5.1.1) provides zero-cost dispatch with full type safety. Be mindful of the exact-match requirement and consider providing `const`-qualified entries.

2. **Use function-pointer tables for extensible interfaces.** When new types may be added without modifying dispatch code, vtable-style approaches offer the best balance of flexibility and performance.

3. **Consider intrusive structures for multiple interfaces.** When objects must participate in several polymorphic abstractions (e.g., multiple lists, trees, or queues), the `container_of` idiom provides zero-overhead composition.

4. **Reserve tagged pointers for memory-critical applications.** The complexity and portability concerns are justified only when per-object overhead is unacceptable.

5. **Respect strict aliasing.** Use union-based access patterns for tagged structures, or document reliance on `-fno-strict-aliasing`.

6. **Document macro complexity.** Signature-based polymorphism through argument counting produces intricate macro expansions; thorough documentation is essential for maintainability.

7 Conclusion

While C lacks built-in polymorphism, the idioms surveyed here, void-pointer tagging, function-pointer tables, tagged pointers, `_Generic`, intrusive structures, and variadic macro techniques, enable construction of flexible, efficient, and reasonably type-safe polymorphic APIs. Each approach entails distinct trade-offs among compile-time safety, runtime cost, memory overhead, and portability. The C11 standard [1] introduced key features (`_Generic`, `aligned_alloc`, `max_align_t`, `_Static_assert`) that significantly enhance the viability of these techniques.

These techniques position C uniquely among systems programming languages: they demand explicit programmer effort but reward that effort with precise control over performance characteristics. As C continues to serve as the foundation for operating systems, embedded systems, and performance-critical libraries, mastery of these idioms remains valuable for practitioners seeking clean, maintainable APIs.

Future work might explore automated generation of polymorphic wrappers through code generation tools, or investigate how C23's `__VA_OPT__` simplifies signature-based dispatch. Additionally, formal verification of type safety properties for these idioms represents an open research direction.

Acknowledgments

The author acknowledges the broader C programming community, whose collective experimentation over decades has refined these techniques. The `container_of` idiom, in particular, owes its widespread adoption to the Linux kernel development community.

References

[1] ISO/IEC 9899:2011, *Programming Languages—C*, International Organization for Standardization, 2011.

[2] S. P. Harbison and G. L. Steele Jr., *C: A Reference Manual*, 5th ed., Prentice Hall, 2002.

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed., O'Reilly Media, 2005.

[4] J. Gustedt, *Modern C*, Manning Publications, 2019.

[5] D. R. Hanson, *C Interfaces and Implementations*, Addison-Wesley, 1996.

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, 1988.

A VRG: A Complete Variadic Macro Library

Section 3 presented basic techniques for signature-based polymorphism through argument counting. However, those techniques share a fundamental limitation: they cannot distinguish between *zero arguments* and *one argument*. This appendix presents VRG, a single-header library that solves this problem through clever preprocessor techniques.

A.1 The Zero-Argument Problem

Standard C preprocessor macros cannot distinguish between zero arguments and one empty argument:

```
#define COUNT(...) /* how many args? */

COUNT()      // Is this 0 args or 1 empty arg?
COUNT(x)     // This is clearly 1 arg
```

Using the basic `ARG_COUNT` macro from Section 3, both `COUNT()` and `COUNT(x)` would return 1. VRG solves this limitation entirely at compile time, with no runtime overhead.

A.2 Quick Start

The following example demonstrates the main `vrg()` macro, which dispatches to different handlers based on argument count:

```
#include "vrg.h"

// Your actual implementation function
int my_func_impl(int a, char b, void *c) {
    // ... implementation ...
}

// Define the variadic interface
#define myfunc(...)       vrg(myfunc_, __VA_ARGS__)
#define myfunc_0()        my_func_impl(0, '\0', NULL)
#define myfunc_1(a)       my_func_impl(a, '\0', NULL)
#define myfunc_2(a, b)    my_func_impl(a, b, NULL)
#define myfunc_3(a, b, c) my_func_impl(a, b, c)

// Now you can call:
myfunc();           // Calls myfunc_0()
myfunc(42);         // Calls myfunc_1(42)
myfunc(42, 'x');    // Calls myfunc_2(42, 'x')
myfunc(42, 'x', p); // Calls myfunc_3(42, 'x', p)
```

For simpler cases where you only need to distinguish zero-vs-many or one-vs-many arguments, use `vrg0()` or `vrg1()`:

```
#include "vrg.h"
#include <stdio.h>

// vrg0: distinguish between zero and any number of arguments
#define debug(...) vrg0(debug_, __VA_ARGS__)
#define debug_0()      printf("[DEBUG] checkpoint\n")
```

```c
#define debug__(...)    printf("[DEBUG] " __VA_ARGS__)

debug();                        // [DEBUG] checkpoint
debug("x = %d\n", 42);          // [DEBUG] x = 42

// vrg1: distinguish between one-or-fewer and more-than-one
#define warn(...) vrg1(warn_, __VA_ARGS__)
#define warn_1(code)    fprintf(stderr, "Warning %d\n", code)
#define warn__(...)     fprintf(stderr, __VA_ARGS__)

warn(404);                      // Warning 404
warn("File %s not found\n", name);  // File ... not found
```

A.3 API Reference

VRG provides several macros for different use cases:

A.3.1 `vrg(prefix, ...)`

The main macro for creating variadic functions with specific handlers for 0–9 arguments. It counts the arguments and appends that count to the prefix, then calls the resulting macro.

```c
#define myfunc(...) vrg(myfunc_, __VA_ARGS__)
// myfunc(a, b) expands to myfunc_2(a, b)
```

When defining handlers, use the following naming convention:

Macro	Called When
`prefix_0()`	Zero arguments
`prefix_1(a)`	One argument
`prefix_2(a,b)`	Two arguments
...	...
`prefix_9(...)`	Nine arguments

A.3.2 `vrg0(prefix, ...)`

Distinguishes between *zero arguments* and *any number of arguments*. Expected handlers:

```c
#define print(...) vrg0(print_, __VA_ARGS__)
#define print_0()       printf("(nothing)\n")
#define print__(...)    printf(__VA_ARGS__)

print();              // printf("(nothing)\n")
print("x=%d\n", x);   // printf("x=%d\n", x)
```

A.3.3 `vrg1(prefix, ...)`

Distinguishes between *one or fewer arguments* and *more than one argument*. Expected handlers:

```
#define log(...) vrg1(log_, __VA_ARGS__)
#define log_1(level)    log_impl(level, "default message")
#define log__(...)      log_impl(__VA_ARGS__)

log(INFO);                      // log_impl(INFO, "default message")
log(ERROR, "Oops: %d", code);  // log_impl(ERROR, "Oops: %d", code)
```

A.3.4 `vrg2(prefix, ...)`

Distinguishes between *two or fewer arguments* and *more than two arguments*.

A.4 Implementation Details

This is the full code for the `vrg` macros:

```
#ifndef VRG_VERSION
#define VRG_VERSION 0x0021000B // 0.21.0-beta
#define VRG_jn(x,y)    VRG_exp(x ## y)
#define VRG_join(x,y)  VRG_jn(x, y)
#define VRG_exp(...) __VA_ARGS__
#define VRG_count(x1,x2,x3,x4,x5,x6,x7,x8,x9,xA,xN, ...) xN
#define VRG_nargs(...)    VRG_exp(VRG_count(__VA_ARGS__, A, 9, 8, 7, 6, 5,
    4, 3, 2, 1, 0))
#define VRG_ncommas(...)  VRG_exp(VRG_count(__VA_ARGS__, _, _, _, _, _, _, _
    , _, 1, _, _))
#define VRG_comma(...) ,
#define VRG_sel(x,...) \
   VRG_join(VRG_sel_, \
            VRG_join(VRG_ncommas(VRG_comma __VA_ARGS__ ()), VRG_ncommas(
    VRG_comma __VA_ARGS__ ))) (x)
#define VRG_sel_1_(x) 0
#define VRG_sel_11(x) x
#define VRG_sel___(x) x
#define vrg(f_,...)  VRG_join(f_, VRG_sel(VRG_nargs(__VA_ARGS__),__VA_ARGS__
    ))(__VA_ARGS__)
#define VRG_frst(x,...) x
#define VRG_scnd(x,...) VRG_frst(__VA_ARGS__)
#define VRG_tail(x,...) __VA_ARGS__
#define VRG_tail2(x,...) VRG_tail(__VA_ARGS__)
#define VRG_precomma(...) VRG_comma
#define VRG_sel_n(x,...) \
```

```
23    VRG_join(VRG_sel_ ,VRG_ncommas(VRG_exp(VRG_precomma VRG_frst(__VA_ARGS__)
        () VRG_scnd(__VA_ARGS__) ()))))(x)
24 #define VRG_sel_1(x) x
25 #define VRG_sel__(x) _
26 #define vrg0(f_,...)  VRG_join(f_,VRG_sel_n(0,__VA_ARGS__))(__VA_ARGS__)
27 #define vrg1(f_,...)  VRG_join(f_,VRG_sel_n(1,VRG_tail(__VA_ARGS__)))(
       __VA_ARGS__)
28 #define vrg2(f_,...)  VRG_join(f_,VRG_sel_n(2,VRG_tail2(__VA_ARGS__)))(
       __VA_ARGS__)
29 #define vrg_(f_,...)  vrg0(f_,...)
30 #endif // VRG_VERSION_H
```

A.4.1 Token Joining

The `VRG_join` macro joins two tokens together. The two-step process ensures macro arguments are fully expanded before concatenation:

```
#define VRG_jn(x, y)   VRG_exp(x ## y)
#define VRG_join(x, y) VRG_jn(x, y)

// VRG_join(foo_, 3) expands to foo_3
```

A.4.2 Extra Expansion

The `VRG_exp` macro forces an extra expansion pass, crucial for Microsoft's MSVC compiler:

```
#define VRG_exp(...) __VA_ARGS__
```

A.4.3 Argument Counting

The core counting mechanism shifts arguments through a fixed-position selector:

```
#define VRG_count(x1,x2,x3,x4,x5,x6,x7,x8,x9,xN,...) xN
#define VRG_nargs(...) \
    VRG_exp(VRG_count(__VA_ARGS__,9,8,7,6,5,4,3,2,1,0))
```

For example, `VRG_nargs(a, b, c)` expands as:

```
VRG_count(a, b, c, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
//        x1 x2 x3 x4 x5 x6 x7 x8 x9 xN
//                                   ^-- returns 3
```

A.4.4 The Zero-Argument Detection Algorithm

This is the heart of VRG. It uses two expressions involving a special `VRG_comma` macro:

```
#define VRG_comma() ,
```

The key insight is that `VRG_comma` by itself does not expand, but `VRG_comma()` expands to a comma. By analyzing two expressions:

1. `VRG_comma __VA_ARGS__ ()`
2. `VRG_comma __VA_ARGS__`

and counting commas in each, VRG can distinguish the cases:

Arguments	Expr1 Commas	Expr2 Commas	Combined
None	1	_	1_
One (with cast) or two	1	1	11
All other cases	_	_	__

A selector macro then uses token pasting to dispatch:

- `VRG_sel_1_()` returns `0` (zero arguments detected)
- `VRG_sel_11()` and `VRG_sel__()` return the actual argument count

A.5 Complete Examples

A.5.1 Optional Arguments with Defaults

```c
#include "vrg.h"
#include <stdio.h>

void greet_impl(const char *name, const char *greeting, int times) {
    for (int i = 0; i < times; i++)
        printf("%s, %s!\n", greeting, name);
}

#define greet(...)      vrg(greet_, __VA_ARGS__)
#define greet_0()       greet_impl("World", "Hello", 1)
#define greet_1(n)      greet_impl(n, "Hello", 1)
#define greet_2(n,g)    greet_impl(n, g, 1)
#define greet_3(n,g,t)  greet_impl(n, g, t)

int main() {
    greet();                    // Hello, World!
    greet("Alice");             // Hello, Alice!
    greet("Bob", "Hi");         // Hi, Bob!
    greet("Carol", "Hey", 3);   // Hey, Carol! (3 times)
}
```

A.5.2 Debug Macro with Optional Message

```c
#include "vrg.h"
#include <stdio.h>

#define debug(...) vrg0(debug_, __VA_ARGS__)
#define debug_0()      printf("[DEBUG] checkpoint reached\n")
#define debug__(...)   printf("[DEBUG] " __VA_ARGS__)

int main() {
    debug();                        // [DEBUG] checkpoint reached
    debug("value = %d\n", 42);      // [DEBUG] value = 42
}
```

A.5.3 Assertion Macro with Optional Message

```c
#include "vrg.h"
#include <stdio.h>
#include <stdlib.h>

void assert_fail(const char *expr, const char *msg,
                 const char *file, int line) {
    fprintf(stderr, "Assertion failed: %s\n", expr);
    if (msg) fprintf(stderr, "Message: %s\n", msg);
    fprintf(stderr, "Location: %s:%d\n", file, line);
    abort();
}

#define my_assert(...) vrg1(my_assert_, __VA_ARGS__)
#define my_assert_1(expr) \
    do { if (!(expr)) assert_fail(#expr, NULL, \
         __FILE__, __LINE__); } while(0)
#define my_assert__(expr, ...) \
    do { if (!(expr)) assert_fail(#expr, __VA_ARGS__, \
         __FILE__, __LINE__); } while(0)

int main() {
    int x = 5;
    my_assert(x > 0);                          // No message
    my_assert(x < 10, "x must be under 10");  // With message
}
```

A.6 Limitations and Compatibility

Limitations:

1. **Maximum 9 arguments**: The counting mechanism supports 0–9 arguments.
2. **Type-cast first arguments**: If the first argument is type-cast like `(int)x`, the parentheses may cause issues with zero-detection. VRG handles common cases, but complex expressions may fail.
3. **Compile-time only**: This is purely preprocessor-based. Runtime variadic argument counts are not supported.
4. **Comma in arguments**: Arguments containing commas need extra parentheses, e.g., `foo((a,b))`.

Compiler Compatibility:

- **C99 and later**: Requires `__VA_ARGS__` support ([1], §6.10.3.1)
- **GCC**: Fully supported
- **Clang**: Fully supported
- **MSVC**: Supported (uses `VRG_exp` for extra expansion passes)

A.7 Availability

The VRG library is available at `https://github.com/rdentato/vrg` under the MIT License.